Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation

Miguel Gómez-Zamalloa[1] Elvira Albert[1] Germán Puebla[2]

[1] *DSIC, Complutense University of Madrid,* {mzamalloa,elvira}@fdi.ucm.es

[2] *Technical University of Madrid,* german@fi.upm.es

Abstract

The *interpretative approach* to compilation allows compiling programs by partially evaluating an interpreter w.r.t. a source program. This approach, though very attractive in principle, has not been widely applied in practice mainly because of the difficulty in finding a partial evaluation strategy which always obtain "quality" compiled programs. In spite of this, in recent work we have performed a proof of concept of that, at least for some examples, this approach can be applied to *decompile* Java bytecode into Prolog. This allows applying existing advanced tools for analysis of logic programs in order to verify Java bytecode. However, successful partial evaluation of an interpreter for (a realistic subset of) Java bytecode is a rather challenging problem. The aim of this work is to improve the performance of the decompilation process above in two respects. First, we would like to obtain quality decompiled programs, i.e., simple and small. We refer to this as the *effectiveness* of the decompilation. Second, we would like the decompilation process to be as efficient as possible, both in terms of time and memory usage, in order to scale up in practice. We refer to this as the *efficiency* of the decompilation. With this aim, we propose several techniques for improving the partial evaluation strategy. We argue that our experimental results show that we are able to improve significantly the efficiency and effectiveness of the decompilation process.

1 Introduction

Partial evaluation [12] is a semantics-based program transformation technique whose main purpose is to optimize programs by specializing them w.r.t. part of their input (the *static* data)—hence it is also known as *program specialization.* Essentially, given a program P and a static data s, a partial evaluator returns a residual program P_s which is a specialized version of P w.r.t. the static data s such that $P(s,d) = P_s(d)$ for all *dynamic* (i.e., not static) data d. The development of partial evaluation techniques [12] has led to the so-called "interpretative approach" to compilation, also known as first Futamura projection [5]. In this approach, compilation of a source program P from a source language $\mathcal{L}_S$ to a target language $\mathcal{L}_O$ can in principle be performed by specializing an interpreter *Int* for $\mathcal{L}_S$ written in $\mathcal{L}_O$ w.r.t. P. The program Int_P thus obtained can be akin to the result $Comp_S^O\ (P)$ of direct compilation of P using a compiler $Comp_S^O$ from $\mathcal{L}_S$ to $\mathcal{L}_O$. When $\mathcal{L}_S$ is Java bytecode and $\mathcal{L}_O$ is Prolog, we theoretically obtain a "decompilation" from (low-level) Java bytecode to (high-level) Prolog programs [1]. The motivation for obtaining a high level logic representation of the Java bytecode is clear: we can apply advanced tools developed for high level languages to the resulting programs without having to deal with the complicated unstructured control flow of the bytecode, the use of the stack, the exception handling, its object-oriented features, etc. In particular, for logic programming, we have available generic analysis tools which are incremental [10] and modular [4] that we will be able to directly use [1]. The motivations for using the interpretative approach to decompilation rather than implementing a compiler from Java bytecode to LP include: 1) flexibility, in the sense that it is easy to modify the interpreter in order to observe new properties of interest, 2) easy of trust, in the sense that it is rather difficult to prove (or trust) that the compiler preserves the program semantics and, it is also complicated to explicitly specify what the semantics used is, 3) easier to maintain, new changes in the JVM semantics can be easily reflected in the interpreter, and 4) easier to implement, provided a powerful partial evaluator for LP is available.

The success of the interpretative approach highly depends on eliminating the overhead of parsing the program, fetching instructions, etc., thus obtaining programs which are akin to those obtained by a traditional compiler. When both the $\mathcal{L}_S$ and $\mathcal{L}_O$ languages are the same, fully getting rid of the layer of interpretation is known as "Jones optimality" [11,12] and intuitively means that the result of specializing an interpreter *Int* w.r.t a program P should be basically the same as P, i.e., $Int_P \approx P$. Specializing interpreters has been a subject of research for many years, especially in the logic programming community (see, e.g., [22,23,15] and their references). However, despite these efforts, achieving Jones optimality in a systematic way is not straightforward since, given a program P, there are an infinite number of residual programs Int_P which can be obtained, and only a small fraction of them are akin to the results of direct compilation. As a result, only partial success has been achieved to date, such as in the specialization of a simple Vanilla interpreter, of the same interpreter extended with a debugger, and of a lambda interpreter [15].

The first requirement for achieving effective decompilation is to have a partial evaluator which is powerful (or "aggressive" in partial evaluation terminology) enough so as to remove the overhead of the interpretation level from the residual program. In a sense, the work in [1] shows that our partial evaluator [20,2] is aggressive enough for being used in the interpretative approach. The next two questions we need to answer, and which are addressed in this work are: is the control strategy used too aggressive in some cases? If so, it is possible to fix this problem? Note that the consequences of the strategy being too aggressive can be rather negative: it can introduce non-termination in the decompilation process and, even if the process terminates, it can result in inefficient decompilation (both in terms of time and memory) and in unnecessarily large residual programs. It should be noted that memory efficiency of the decompilation process is quite important since it can happen that the decompiler fails to generate a residual program because the partial evaluator runs out of memory.

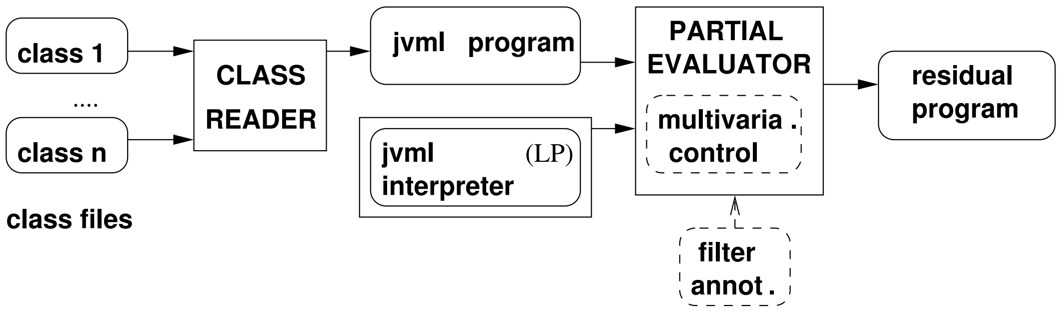

Fig. 1. Decompilation of Java Bytecode into Prolog by online PE w/ offline annotations

2 An Overview of the Decompilation Process

Figure 1 shows an overview of the interpretative decompilation process originally proposed in [1] and followed in this paper. Initially, given a set of `.class` files {`class 1`,..., `class n`}, a program called *class reader*, returns a representation of them in `Ciao` Prolog [3]. We use a slightly modified JVM language where some bytecode instructions are factorized and which contains some other minor simplifications (see [1]). Then, we have a JVML interpreter written in `Ciao` which captures the JVM semantics. The decompilation process consists in specializing the JVML interpreter w.r.t. the LP representation of the classes. In this work, we will improve the decompilation by introducing two new elements (which appear within a dashed box in the figure): an improved *multi-variance* control within the partial evaluator and *filter annotations* to refine the control of the *partial evaluator*.

2.1 The LP Representation of the Bytecode

The LP (`Ciao`) program generated by the *class reader* contains the bytecode instructions for all methods in {`class 1`,..., `class n`}. They are represented as a set of facts `bytecode`; and also, a single fact `class` obtained by putting together all the other information available in the `.class` files (class name, methods and fields signatures, etc.). Each `bytecode` fact is of the form `bytecode(PC,MethodID,Class,Inst, Size)`, where `Class` and `MethodID`, respectively, identify the class and the method to which the instruction `Inst` belongs. `PC` corresponds to the program counter and `Size` to the number of bytes of the instruction in order to be able to compute the next value of the program counter. The form of the fact `class` is not relevant to this work but it can be observed in [1].

Example 2.1 [LP representation] Our running example consists of the single Java class `LinearSearch`, which appears in Fig 2. To the right, we show the `bytecode` facts corresponding to the method `search` identified with number "0" (second argument) of class number "1" (third argument). Bytecodes labeled from 0 to 6 (first argument) correspond to the first three initialization instructions in the Java program. Then, if the first conjunct in the `while` condition does not hold (bytecodes 8-11), the PC moves 26 positions downwards (i.e., to bytecode 37). Otherwise, the second conjunct is checked and similarly the PC can be increased in 22 positions (i.e., to bytecode 37). The condition in the `if` instruction corresponds to bytecodes

```
class LinearSearch{
 static int search(int[] xs,int x){
    int size = xs.length;
    boolean found = false;
    int i = 0;
    while ((i<size)&&(!found)){
      if (xs[i] == x) found = true;
      else i++;
    }
    return i;
 }
```

```
bytecode(0,'0',1,aload(0),1).
bytecode(1,'0',1,arraylength,1).
bytecode(2,'0',1,istore(2),1).
bytecode(3,'0',1,const(
          primitiveType(int),0),1).
bytecode(4,'0',1,istore(3),1).
bytecode(5,'0',1,const(
          primitiveType(int),0),1).
bytecode(6,'0',1,istore(4),2).
bytecode(8,'0',1,iload(4),2).
bytecode(10,'0',1,iload(2),1).
bytecode(11,'0',1,if_icmp(geInt,26),3).
bytecode(14,'0',1,iload(3),1).
bytecode(15,'0',1,if0(neInt,22),3).
bytecode(18,'0',1,aload(0),1).
bytecode(19,'0',1,iload(4),2).
bytecode(21,'0',1,iaload,1).
bytecode(22,'0',1,iload(1),1).
bytecode(23,'0',1,if_icmp(neInt,8),3).
bytecode(26,'0',1,const(
          primitiveType(int),1),1).
bytecode(27,'0',1,istore(3),1).
bytecode(28,'0',1,goto(-20),3).
bytecode(31,'0',1,iinc(4,1),3).
bytecode(34,'0',1,goto(-26),3).
bytecode(37,'0',1,iload(4),2).
bytecode(39,'0',1,ireturn,1).
```

Fig. 2. Java code and *LP* representation of Running Example

18-23, the `then` branch to 26-28 and the `else` branch to 31-34. Finally, bytecodes 37-39 represent the return.

2.2 The JVML Interpreter

The JVML interpreter expresses the JVM semantics in `Ciao` following the formal specification in Bicolano [19]. In our specification, a *state* is modeled by a term of the form $st(Heap, Frame, StackFrame)$ which represents the machine's state where: $Heap$ represents the contents of the heap, $Frame$ represents the execution state of the current $Method$ and $StackFrame$ is a list of frames corresponding to the call stack. Each frame is of the form $fr(Method, PC, OperandStack, LocalVar)$ and contains the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point PC of the method $Method$. Note that, whenever we are at an exception state, the state and the frames will be represented accordingly as stE and frE terms resp., with the same arguments as their homologous st and fr, except for the $OperandStack$ which will be a location number (instead of a list) referencing the corresponding exception object in the heap.

Fig. 3 shows a fragment of the `Ciao` JVML interpreter. Given the program and the current state, its main predicate `execute` first calls predicate `step`, which produces the *state* after executing the corresponding bytecode. The process iterates with a recursive call to predicate `execute` with the new state until one of the following conditions holds: 1) we reach a return instruction (i.e. `return`, `ireturn` or `areturn`), with the JVM call stack being empty, 2) we are in an exception state for which no suitable exception handler has been found, with the JVM call stack being empty, 3) there is no bytecode instruction at the current `PC`. The latter should never occur for a valid bytecode program. The whole interpreter, together with a collection of examples, are available at: `http://cliplab.org/Systems/jvm-by-pe`.

```
execute(Program,State,FinalState) :-
        step(_,Program,State,NextState),
        execute(Program,NextState,FinalState).
execute(_P,State,State) :-
        check_return(State).
execute(Program,State,NextState) :-
        State=stE(Heap,frE(Method,PC,Loc,_),[]),
        NextState=st(Heap,fr(Method,PC,[ref(Loc)],_),[]),
        not_handled_exception(Program,State).

check_return(st(_H,fr(Method,PC,_Stack,_L),[])) :-
        instructionAt(Method,PC,return).
check_return(st(_H,fr(Method,PC,[num(int(_I))|_Stack],_L),[])) :-
        instructionAt(Method,PC,ireturn).
check_return(st(_H,fr(Method,PC,[ref(loc(_I))|_Stack],_L),[])) :-
        instructionAt(Method,PC,areturn).

step(goto_step_ok,_P,st(H,fr(M,PC,S,L),SF),st(H,fr(M,PCb,S,L),SF)) :-
        instructionAt(M,PC,goto(O)),
        PCb is PC+O.
...
```

Fig. 3. Fragment of the JVML interpreter

3 Basics of Online Partial Evaluation of Logic Programs

We assume familiarity with basic notions of logic programming [18]. Executing a logic program P for an atom A consists in building a so-called SLD tree for $P \cup \{A\}$ and then extracting the computed answer substitutions from every non-failing branch of the tree. Online partial evaluation builds upon the execution approach of logic programs with two main differences:

- In order to guarantee termination of the *unfolding* process, when building the SLD-trees, it is possible to choose *not* to further unfold a goal, and rather leave a leaf in the tree with a non-empty, possibly non-failing, goal. The resulting SLD is called a *partial* SLD tree. Note that even if the SLD trees for all possible queries are finite, the SLD to be built during partial evaluation may be infinite. The reason for this is that since dynamic values are not known at specialization time, the specialization SLD tree can have more branches (in particular, infinite branches) than the actual SLD tree at run-time. Which atom to select from each resolvent and when to stop unfolding is determined by the *unfolding rule*.
- The partial evaluator may have to build several SLD-trees to ensure that all atoms left in the leaves are "covered" by the root of some tree (this is known as the closeness condition of partial evaluation [17]). The so-called *abstraction operator* performs "generalizations" on the atoms that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of atoms. When all atoms are covered, then there is no need to build more trees and the process finishes. Details on abstraction operators appear in Section 4.

The essence of most algorithms for on-line partial evaluation of logic programs (see e.g. [8]) can be viewed in the algorithm shown in Figure 4, which is parametric w.r.t. the unfolding rule, unfold, and the abstraction operator, abstract. It starts from a program P and an initial set of atoms S. At each iteration, the *local control* is performed by the unfold rule which takes the current set of atoms S_i and the program and constructs partial SLD trees for the atoms in S_i. In the *global control*, when some calls in the leaves of the trees (named $\mathcal{T}_{calls}$ in the algorithm) are not

```
Input: a program $P$ and a set of atoms $S$
Output: a set of atoms $T$
 Initialization: $i := 0$;  $S_0 := S$
 Repeat
         1.   $\mathcal{T} := \mathsf{unfold}(S_i, P)$;
         2.   $S_{i+1} := \mathsf{abstract}(S_i, \mathcal{T}_{calls})$;
         3.   $i := i + 1$;
 Until $S_i = S_{i-1}$ (modulo renaming)
 Return $T := S_i$
```

Fig. 4. Partial Evaluation Algorithm

properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached). Thus, basically, the algorithm iteratively constructs partial SLD trees until all their leaves are covered by the root nodes.

A partial evaluation of P w.r.t. S can then be systematically extracted from the resulting set of atoms T. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD-trees for the final set of atoms T. In particular, given an SLD derivation of $P \cup \{A\}$ with $A \in T$ ending in B and θ the composition of the mgu's in the derivation step, then the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A *partial evaluation* is then defined as the sequence of resultants associated to the derivations of the constructed partial SLD trees for all $P \cup \{A\}$ with $A \in T$.

4 Challenges in Specialization of JVM Interpreter

In order to achieve an *effective* decompilation, one of the crucial requirements is to have available control strategies (i.e., unfold and abstract operators) which are powerful enough to remove the interpreter overhead. For this reason, the experiments in [1] have been performed by using "aggressive" control strategies based on *homeomorphic embedding* [13,14]. In local control, by aggressiveness we mean unfolding rules which compute derivations as long as possible provided there are no termination problems. In global control, it denotes abstraction operators which generalize in as few situations as possible without endangering termination.

4.1 A Challenging Example

The example in Fig. 5 is instrumental to show the challenges which appear in the specialization of the JVM interpreter in Section 2.2. The specialization process starts by running the PE algorithm of Section 3 for the initial program P being the JVM interpreter and the following initial atom:

```
execute(Prog,st(heap([array(locationArray(_,primitiveType(int)),_)]),
                 fr(method('int LinearSearch.search(int[],int)'),
                    0,[],[ref(1),_,0,0,0]),
                []),_)
```

where "Prog" would be instantiated to the constant term representing the corresponding JVM program of Sect. 2.1, and a "_" represents a logical variable. Let us

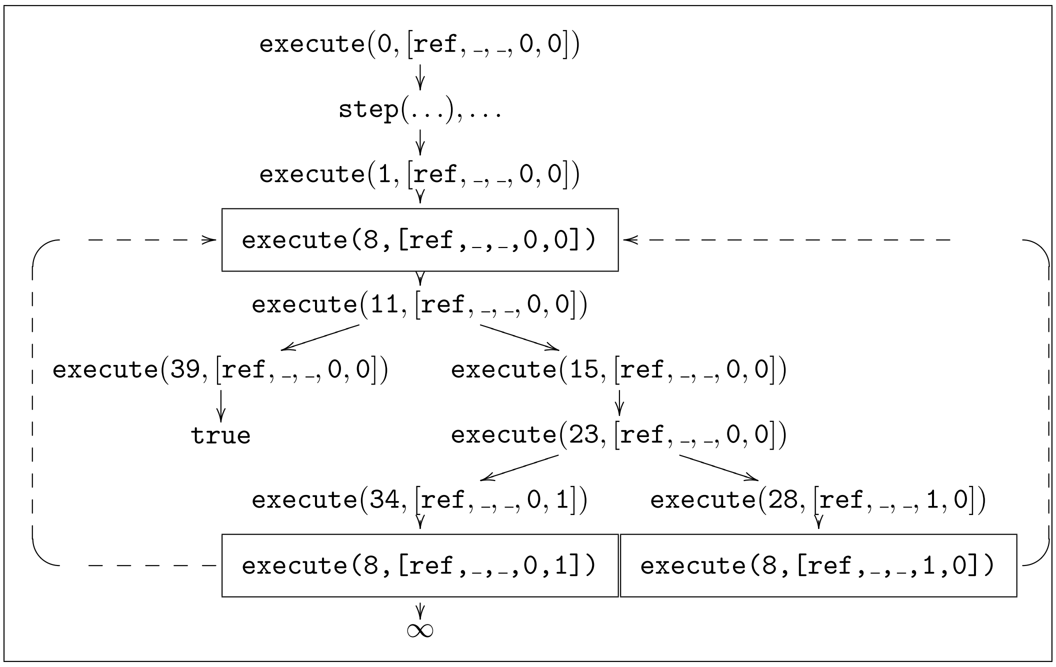
execute(0, [ref, _, _, 0, 0])
step(...), ...
execute(1, [ref, _, _, 0, 0])
execute(8, [ref, _, _, 0, 0])
execute(11, [ref, _, _, 0, 0])
execute(39, [ref, _, _, 0, 0])
true
execute(15, [ref, _, _, 0, 0])
execute(23, [ref, _, _, 0, 0])
execute(34, [ref, _, _, 0, 1])
execute(28, [ref, _, _, 1, 0])
execute(8, [ref, _, _, 0, 1])
execute(8, [ref, _, _, 1, 0])
∞

Fig. 5. Partial SLD Tree of Specialization of JVM Interpreter

note that this initial state has been built from a "method invocation specification" (MIS), i.e., a high level description specifying the method we want to decompile and its arguments values. In our case, we want to decompile a method for computing a linear search for any array of integers and any value as argument. Thus, we use "`int LinearSearch.search(int[] _,int _)`"as MIS.

In the figure, we depict (a reduced version of) one of the SLD trees that lead to an effective decompilation of our running example. In order to focus the attention to the relevant arguments only, each atom of the form `execute`(`Program`, `st`(`Heap`, `fr`(`Method`, `PC`, `Stack`, `LocalVar`), `CallStack`), `FinalState`) is represented in the figure as `execute`(`PC`, `LocalVar`) to show only its two key arguments. Indeed, the argument `Program` and `Method` are always constants, the `Stack` is not relevant and the `Heap` is not used in this example. The `CallStack` is always the empty list since the considered method does not invoke any other method (nor itself) and `FinalState` is always a fresh variable. Another simplification in the figure is that each arrow involves the application of several unfolding steps. In particular, the execution of the `step` predicate can be considered as a black box during unfolding, in the sense that it performs all the operations (i.e., a number of unfolding steps) and returns the corresponding state. Therefore, we can ignore the intermediate steps produced in order to unfold the calls to `step` and view each of the derivations as a sequence of the form `execute, step, execute, step,` ... (in the figure actually we only show one `step`). Some of the statements within the body of each `step` operation can stay as residual when they involve data which is not known at specialization time. The computation rule during unfolding is able to residualize calls which are not-sufficiently instantiated and select non-leftmost atoms in a safe way [2], in particular, further calls to `execute`.

4.2 Control Strategies based on Embedding

The interested reader is referred to Leuschel's work [16] where a detailed description of the embedding relation can be found. Informally, atom t_1 *embeds* atom t_2, written $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g., $\mathtt{s}(\underline{\mathtt{s}}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}+}\mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$. By relying on the embedding relation, the following strategies can be defined (they correspond to those used in [1]):

4.2.1 Local Control

Unfolding operators based on the homeomorphic embedding $\trianglelefteq$, denoted $\mathsf{unfold}_{\trianglelefteq}$, allow the expansion of derivations until reaching an atom which *embeds* some of the previous atoms in its sequence of covering ancestors (see e.g., [20]). The intuition is that reaching larger (or equal) atoms in the same derivation can endanger termination and hence the computation has to be stopped. Furthermore, in order to achieve the required level of aggressiveness it is also required to be able to accurately handle builtin predicates and to safely perform non-leftmost unfolding [2]. However, in the presence of an infinite signature (e.g., integers) as we have in the JVM interpreter, this unfolding rule can lead to non-terminating computations. Consider, for example, a sequence of atoms of the form: $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,0])$, $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,1])$, $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,2])\ldots$, which can grow infinitely and which the homeomorphic embedding does not flag as potentially dangerous. As a result, by considering the usual homeomorphic embedding relation, the second branch of the partial SLD in Figure 5 is not flagged as dangerous and unfolding does not terminate. This is indicated in the figure by the ∞ symbol as continuation of the second branch. A possible relatively straightforward solution for avoiding this nonterminating behavior of unfolding is to use a slight adaptation of the original homeomorphic relation in which any number embeds any other number, denoted $\trianglelefteq_{num}$. Under this relation the atom $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,1])$ embeds $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,0])$ (and vice-versa). Unfortunately, this modification to the homeomorphic embedding relation, although it guarantees termination of the partial evaluation process is a too coarse approximation and leads to excessive precision loss. It turns out not to be an acceptable alternative for specialization of our interpreter since in virtually all cases the residual program contains the full interpreter, i.e., we have not been able to eliminate the interpretation layer.

4.2.2 Global Control

The homeomorphic embedding ordering $\trianglelefteq$ can also be used at the global control level within the abstract operator $\mathsf{abstract}_{\trianglelefteq}$ in order to decide when to generalize (i.e., to apply the *most specific generalization*) before proceeding to build (possibly partial) SLD trees. Basically, for each new atom A, it checks whether it is larger than (i.e., it embeds) any of the atoms in the set S_i (which contains the atoms in the roots of the partial trees which have already been built). If A does not embed any atom in S_i, it is added to the set; otherwise, the two atoms are generalized by using the *msg* operator. For instance, if we have $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,0])$ in S_i and we want to add the atom $\mathtt{execute}(8,[\mathtt{ref},\_,\_,1,0])$, by using the original homeomorphic embedding relation, no danger is flagged. Thus, in order to guarantee termination at the global control level we also need to modify the relation to be used when infinite signatures (numbers) are considered. By using the modified embedding relation with numbers $\trianglelefteq_{num}$, the latter atom is generalized into $\texttt{execute}(8,[\texttt{ref},\_,\_,\texttt{X},0])$ before being introduced in S_i.

Regarding the *efficiency* of the PE process, it should be noted that the use of control strategies based on embedding introduces a significant overhead, as we need to keep track of the ancestors (see, e.g., [20]) and to perform expensive embedding checks for each of the atom arguments.

5 Partial Evaluation Types for Decompilation

As we have seen in the previous section, in the presence of an infinite signature, like the integers, neither $\trianglelefteq$ nor $\trianglelefteq_{num}$ alone can achieve effective and efficient decompilations. In particular, the use of "$\trianglelefteq$" can be too aggressive in the sense that it leads to too long derivations (even endangering termination), which prevents from a quality decompilation. In contrast, the use of "$\trianglelefteq_{num}$" is definitely too conservative in the sense that stops derivations too early, which causes the loss of essential information to get a quality decompiled program. In this section, we propose to use the *partial evaluation types* of [9] in order to provide additional information to the PE process and improve the results achieved by using previous techniques based on the above embedding orderings. Such additional information is program-dependent and thus, it makes sense to compute it when we are interested in repeatedly partially evaluating a program. This is obviously the case in our approach to decompilation, since we are repeatedly specializing the interpreter w.r.t. different bytecode programs. This information is provided by means of optional partial evaluation types as defined in [9]. They will allow us to give a selective, context-dependent treatment to arguments at PE time. In particular, the following *basic types* are distinguished:

- dyn: which stands for *dynamic*. This type is used to avoid too aggressive strategies. It denotes that the user thinks it is a good idea to *lose* the information stored in the corresponding argument as soon as a discrepancy is found w.r.t. another "similar" atom. Note that unless such information is lost, increased polyvariance is required in order to maintain separate call patterns with the corresponding values of the discrepant information. This results in higher specialization cost and in a larger residual program. An example of an argument which can be marked as *dynamic* is `Loc` (local variables) in our running example.
- f_sig: which stands for *finite signature*. Literally, this means that the number of functors and constant names which may appear is finite. Thus, for arguments of this type, $\trianglelefteq$ guarantees termination. The motivation for considering this type is that it avoids the need for using $\trianglelefteq_{num}$ for arguments which may contain numbers. The user can use this type for those arguments which are guaranteed to contain a *finite* set of numbers only. This is the case, for instance, of the argument `PC` in our example. Though it is natural to use numbers to represent program counters, given a fixed program, the set of instructions is fixed and finite. This is a key observation which is required to obtain the results presented in this paper.

- const: which stands for *constant*. The motivation for introducing this type is just efficiency of the specialization process. Of course, it should only be applied to arguments which we know will always be instantiated to the same value during specialization time. Its usage does not affect the control strategy at all, but it allows avoiding testing the embedding relation over and over again on arguments which never change. This is the case, for instance, of the argument `Program` which remains constant all over the decompilation process.
- term: which stands for *term*. This the the most general type which includes all possible terms, including partially instantiated terms. This is the default type which is assumed unless the user explicitly provides a more precise pe_type. For programs containing arithmetic (such as our JVM interpreter), the default embedding relation we use is $\trianglelefteq_{num}$ since otherwise termination is not guaranteed.

In order to allow the use of the above basic types at any depth within arguments and, also, allow the possibility of having *disjunctive* types with distinctive functors for which we can declare different types, the notion of *partial evaluation types*, pe_type, is defined [9] as a *regular type* [6] combined with the above basic types.

Let us explain the intuition behind the above pe_type's. The first argument of `execute` is `Program`, which is clearly constant because during each partial evaluation there is exactly one fixed program and there is no need to ever generalize this argument. The third argument is the final (output) `State` which is always a variable before the call and thus it can be given the type term. The type of the current `State` in the second argument is disjunctive and we declare it by means of two rules, one for each functor. The first one corresponds to a normal state `st` and the second one to an exception state `stE`. The most relevant points to note are: 1) The types of the heap and the call stack are declared as dyn as we do not mind "losing" all information about them during partial evaluation when decompiling a method if needed. Intuitively, this is to say that we do not want to generate multiple decompiled versions of a method depending on the state of the heap or the call stack. Instead, as it happens in standard compilation, the decompilation of the method should be independent from the context from which it is called (and hence this information should be ignored). 2) Again, we distinguish two types of `Frames` for normal (`fr`) and exception behavior (`frE`). The important point here is that both the `PC` and `Method` can be instantiated only to a finite number of values, since given a fixed program, the number of methods and the number of different program counters is finite. Therefore, they can be safely declared as f_sig, which prevents from important information loss. Finally, we declare the set of local variables `Loc` and stack positions `Stack` as dyn as they threaten termination as we have seen in the example. Note that termination of the partial evaluation requires that the pe_type's provided are safe. For this it is required that any sub(argument) marked as f_sig actually has a finite signature.

The importance of pe_type declarations is that they can be used at PE time to disregard, to filter or to keep the information available in each argument, as explained above. The embedding relation which makes use of pe_type declarations is called *embedding relation with* pe_type*'s* and written as $\trianglelefteq_{pt}$ [9]. As the traditional embedding relation, it is used to steer the PE process both at the local and global control by means of the corresponding $\mathsf{unfold}_{\trianglelefteq_{pt}}$ and $\mathsf{abstract}_{\trianglelefteq_{pt}}$ operators, respectively.

6 Reducing Polyvariance in Global Control

In the previous section we have seen how the use of suitable partial evaluation types allows keeping the termination guarantees of $\trianglelefteq_{num}$, both at the local and global control levels, while at the same time being aggressive enough so as to get rid of the interpretation layer.

However, though the decompiled programs thus obtained are acceptable, careful inspection of such residual programs shows that relatively often, *useless specialization* has been performed. At the local control level, performing more unfolding than necessary often results in residual predicates defined by many clauses. At the global control level, trying to be too precise results in producing too many predicates in the residual program.

The question is whether there is any way to take the previous *generalization history* into account when abstracting an atom at the global control. The intuition is to keep track of the information which we have been forced to *forget* during the partial evaluation process and proceed to forget it straight away for all new atoms which are *similar* to the previously handled ones under some criteria. The motivation for doing so is that since it seems likely that we will end up being forced to forgetting such info, the sooner we forget such info, the better, both in terms of specialization times and size of the residual program. We now propose a technique based on the ideas above, which can be included inside the standard partial evaluation algorithm, by means of an improved abstraction operator. In order to do that, first, we need to give some preliminary definitions.

A term T is a *generalization* of S (or S is an *instance* of T), denoted by $T \leq S$, if $\exists \sigma.\ T\sigma = S$. Two terms T and T' are *variants*, denoted $T \equiv T'$, if both $T \leq T'$ and $T' \leq T$. If T and T' are variants then there exists a renaming ρ such that $T\rho = T'$. A *generalization* of a set of terms $\{T_1, \ldots, T_n\}$ is another term T such that $\exists \sigma_1, \ldots, \sigma_n$ with $T_i = T\sigma_i$, $i = 1, \ldots, n$. A generalization T is the *most specific generalization* (*msg*) of $\{T_1, \ldots, T_n\}$ if for every other term T' s.t. T' is a generalization of $\{T_1, \ldots, T_n\}$, $T' \leq T$. We also say that two atoms are *homologous*, written as $A \approx B$, if $\mathsf{filter}(A, \mathsf{pe\_type}_A) \equiv \mathsf{filter}(B, \mathsf{pe\_type}_B)$.

Definition 6.1 [HintsTable] We define a HintsTable as a set of pairs of atoms $\langle A, G \rangle$, s.t. $G \leq A$ (i.e., G is a generalization of A).

We refer to these pairs of atoms as *hints* because they provide suggestions on how to *forget* useless information during the abstraction performed at the global control level. Next, we need to define a set of operations over the HintsTable, which will be used later throughout the partial evaluation algorithm both to add and to recover information from the table.

- $\mathsf{addHint}$: HintsTable $\times \langle Atom, Atom \rangle \rightarrow$ HintsTable

$$\mathsf{addHint}(HT, \langle A, G \rangle) = HT \cup \langle A, G \rangle$$

- $\mathsf{applyHint}_{\equiv}$: $\textsc{HintsTable} \times Atom \rightarrow Atom$
  $$\mathsf{applyHint}_{\equiv}(HT, A) = \mathsf{msg}(Gs \cup A)$$
  $$\quad where\ Gs = \{G \mid \langle B, G\rangle \in \textsc{HintsTable}, A \equiv B\}$$
- $\mathsf{applyHint}_{\approx}$: $\textsc{HintsTable} \times Atom \rightarrow Atom$
  $$\mathsf{applyHint}_{\approx}(HT, A) = \mathsf{msg}(Gs \cup A)$$
  $$\quad where\ Gs = \{G \mid \langle B, G\rangle \in \textsc{HintsTable}, A \approx B\}$$

Now, we can define the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ operator by relying on the definitions and operators given above.

Definition 6.2 [$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$] The abstraction operator $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ is defined in terms of the $\mathsf{abstract}_{\trianglelefteq_{pt}}$ operator as follows:

$$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}(S_i, \mathcal{T}_{calls}, HT) = \mathsf{abstract}_{\trianglelefteq_{pt}}(S_i, \mathcal{AT}_{calls})$$
$$where\ \mathcal{AT}_{calls} = \{H \mid H = \mathsf{applyHint}_{\circledcirc}(HT, A),\ \forall A \in \mathcal{T}_{calls},\ \circledcirc \in \{\equiv, \approx\}\}$$

Let us note that the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ operator definition is parametric w.r.t. "$\circledcirc$", and it represents two different abstraction operators, namely $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$ and $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$, depending on which $\mathsf{applyHint}$ operator to use.

After discussing how hints-tables can be exploited during global control, the main question is how exactly we populate such table with the required entries. We propose to simply instrument the $\trianglelefteq_{pt}$ test during partial evaluation in such a way that whenever it flags possible problems between two atoms A and B, i.e., if the relation $A \trianglelefteq_{pt} B$ holds, in addition to returning the value *true*, it also stores the pair $\langle A, msg(A, B)\rangle$ into the hints-table.

Example 6.3 Now, let us consider again the SLD tree in Fig. 5. We start with an empty table of hints $HT = \{\}$. First, in the middle branch, once we reach the embedded atom `execute`$(8, [$`ref`$, \_, \_, 0, 1])$, a new hint will be added to the table by making a call to the $\mathsf{addHint}$ operator. Similarly, another hint will be added in the right branch. Thus, after building the first unfolding tree, the table has the following two entries:

$$HT = \left\{ \begin{array}{l} \langle execute(8, [ref, \_, \_, 0, 1]), execute(8, [ref, \_, \_, 0, Y])\rangle, \\ \langle execute(8, [ref, \_, \_, 1, 0]), execute(8, [ref, \_, \_, X, 0])\rangle \end{array} \right\}$$

Once the unfolding process has finished (see the partial evaluation algorithm in section 3) the following call to the $\mathsf{abstract}$ operator will be made:

$$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}(\{\}, \{execute(8, [ref, \_, \_, 0, 1]), execute(8, [ref, \_, \_, 1, 0])\}, HT)$$

Now, let us explain the effects of the application of each of the different $\mathsf{abstract}$ operators:

- Using $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$. The $\mathsf{applyHint}_{\equiv}$ operator simply returns the corresponding generalized version for each of the atoms. Thus, the standard $\mathsf{abstract}$ operator will be called with $\mathsf{abstract}_{\trianglelefteq_{pt}}(\{\}, \{execute(8, [ref, \_, \_, 0, Y]), execute(8, [ref, \_, \_, X, 0])\})$. Note that, although we keep the same number of different atoms, polyvariance has been potentially reduced as we have generalized a numeric argument, avoiding the possibility of appearing new different versions of the same atom with different numeric values in the corresponding argument.

```
main([[ref(loc(1)),num(int(_))],heap([array(B,A)])],[num(int(0))]) :- 0>=B.
main([[ref(loc(1)),num(int(A))],heap([array(B,[num(int(D))|C])])],[E]) :-
        0<B, D\=A, execute([num(int(D))|C],B,A,0,1,F,E).
main([[ref(loc(1)),num(int(A))],heap([array(B,[num(int(A))|C])])],[D]) :-
        0<B, execute([num(int(A))|C],B,A,1,0,E,D).
main([[null,num(int(_))],heap([])],[ref(loc(1))]).

execute(A,B,C,D,E,heap([array(B,A)]),num(int(E))) :- E>=B.
execute(A,B,C,D,E,heap([array(B,A)]),num(int(E))) :- E<B, D\=0.
execute(A,B,C,0,D,E,J) :-
        D<B, 0=<D, L is D+1, nth(L,A,num(int(M))), M\=C,
        N is D+1, execute(A,B,C,0,N,E,J).
execute(A,B,C,0,D,E,J) :-
        D<B, 0=<D, L is D+1, nth(L,A,num(int(C))),
        execute(A,B,C,1,D,E,J).
```

Fig. 6. Decompiled version of the linear search program

- Using $\mathsf{abstract}_{\unlhd_{pt}+gen_{\approx}}$. In this case, polyvariance will be immediately reduced since, as we will see, both atoms will collapse into the same generalized version. This is due to the generalizations between homologous atoms performed inside the $\mathsf{applyHint}_{\approx}$ operator, which will give rise to the following call to the standard abstract operator $\mathsf{abstract}_{\unlhd_{pt}}(\{\}, \{execute(8, [ref, \_, \_, X, Y]), execute(8, [ref, \_, \_, X, Y])\})$

In Fig. 6 we can see the residual code we have obtained taking advantage of the newly introduced techniques, by partial evaluating the JVML interpreter w.r.t. the bytecode program of our running example (see Fig. 2). Thus, we have used the $\unlhd_{pt}$ as embedding relation (instrumented to add hints when embedding is flagged) and the $\mathsf{abstract}_{\unlhd_{pt}+gen_{\approx}}$ operator. Note that the entry call is `main(In,Out)`, where `In` will be instantiated to the list of argument values specified for the method, together with the input heap, and `Out` will be instantiated to the top of the stack at the end of the execution. This `main` predicate is responsible for first obtaining the initial state and the JVML program and then calling for the first time to the `execute` predicate of the interpreter (represented in the SLD tree in Fig. 5).

In the residual code, we see four rules for predicate `main`, three of them correspond to the three branches represented in the SLD tree, and the fourth one represents the trivial case where the input array is `null` (which, for simplicity, is not represented in the SLD tree). As it can be seen, we have successfully got rid of the interpretation layer as we only have calls to: 1) arithmetic builtins, 2) list builtins (`nth` in this case for accessing the contents of the array) and 3) recursive calls to the `execute` predicate, which represents, in essence, recursive calls to the basic blocks in the control flow graph of the bytecode program.

7 Experimental Results

Table 1 shows the benefits that we can obtain by using pe_type's. We use a set of classical algorithms as benchmarks. We have benchmarks belonging to iterative programs without object-oriented features, thus, **exp**, **gcd**, **lcm** and **fib** compute respectively the exponential, greatest-common-divisor, least-common-multiple and Fibonacci; while **combNoRep**, **CombRep** and **perm** are methods for computing different combinatorial functions. Also, we have some benchmarks using integer arrays, such as **linearSearch** and **binarySearch** which implement the classic linear and binary search over an array; and **Signs** which given an integer array, computes

Benchmark		$\lhd$				$\lhd_{pt}$				**Gains**	
Name	**Size**	**Tm**	**Mem**	**Unf/Eval**	**Size**	**Tm**	**Mem**	**Unf/Eval**	**Size**	**Tm**	**Size**
exp	0.33	1.56	712	1393/227	0.96	0.63	547	1092/187	0.78	2.49	1.23
gcd	0.27	1.19	566	1118/144	0.79	0.48	329	837/110	0.62	2.48	1.26
lcm	0.61	4.15	969	3211/367	2.50	1.39	471	2509/297	2.28	2.98	1.09
combNR	0.33	3.34	1332	2179/287	2.00	0.92	729	1623/216	1.47	3.64	1.36
combR	0.39	5.82	1733	2750/285	2.45	1.47	1203	2131/227	1.78	3.95	1.38
perm	0.28	1.52	562	1099/148	0.85	0.60	321	818/114	0.68	2.53	1.25
add	0.80	29.75	5980	9083/1115	23.15	7.03	3823	6757/830	18.18	4.23	1.27
exp	0.41	8.44	2027	3570/559	4.57	1.22	1079	2444/382	3.16	6.92	1.45
simplify	0.70	14.60	3076	6205/897	8.70	2.87	1917	4774/697	7.26	5.08	1.20
binarySrch	0.42	38.80	9867	10740/1571	29.91	6.00	3361	4837/727	11.53	6.46	2.59
forward	0.60	62.87	4106	14714/2256	16.30	9.20	4108	14714/2256	16.30	6.83	1.00
fib	0.28	—	—	–/–	—	0.64	338	1421/191	1.10	∞	∞
linearSrch	0.32	—	—	–/–	—	1.80	478	2610/394	16.09	∞	∞
signs	0.33	—	—	–/–	—	3.98	1052	4401/702	11.40	∞	∞

Table 1
Measuring the effects of the pe_types

the number of pairs of numbers with different sign. Finally, we have used four benchmarks which make extensive use of object-oriented features such as instance method invocation, field accessing and setting, object creation and initialization, etc. Thus, **add**, **exp** and **simp** compute different operations over rational numbers (represented as objects), while **forward** is invoked over an object representing a date and forwards one day.

For each benchmark, the column **Name** shows the name of the method which is the starting point for the decompilation, and the column **Size** shows its size. All sizes are in KBytes and execution times in seconds. The next four columns, labeled $\unlhd$, provide information about specialization using the original homeomorphic embedding. The first three of them show some data about the specialization process, whereas the fourth one shows the **Size** of the residual program. The aspects which have been measured for the specialization process are **Tm**, which is the time required by partial evaluation, **Mem** which is its memory consumption, and **Unf/Eval** which shows the number of derivation steps together with the number of evaluations steps (i.e., where an `eval` assertion has been applied, see[20]) performed during the partial evaluation process. Similarly, the next four columns provide information about specialization using our proposed combination of embedding with pe_type's. Finally, the last two columns show the gains (in terms of time and size) we obtain with the new embedding definition $\unlhd_{pt}$ based on pe_type's and it is computed as *Old-Cost/New-Cost*. The last three benchmarks do not present data for the $\unlhd$ columns because the partial evaluation process does not terminate for them. As it can be seen in the table, our proposed $\unlhd_{pt}$ specialization is able to handle them. It can also be seen that for all other programs, the use of $\unlhd_{pt}$ results in important gains both in terms of time and size. In terms of time, they range from 2.49 in `exp` to 6.83 times faster in the case of `forward`. The gains in terms of size range from obtaining a similar sized program in `forward` to a program 2.59 times smaller in the case of `binarySearch`.

Benchmark	$abstract_{\lhd_{pt}+gen_{\equiv}}$				$abstract_{\lhd_{pt}+gen_{\approx}}$			
Name	**Tm**	**Mem**	**Unf/Eval**	**Size**	**Tm**	**Mem**	**Unf/Eval**	**Size**
lcm	1.38	1.00	1.46/1.43	1.79	1.41	1.00	1.46/1.43	1.79
add	1.50	1.00	1.56/1.56	1.42	1.25	1.00	1.56/1.56	1.42
simplify	1.37	1.00	1.47/1.46	1.44	1.35	1.00	1.47/1.46	1.44
binarySearch	0.99	1.00	1.00/1.00	1.00	1.10	1.00	1.26/1.22	1.24
linearSearch	1.25	0.98	1.28/1.28	1.11	1.49	0.98	1.80/1.81	4.58
signs	1.24	1.00	1.30/1.30	1.28	1.86	1.00	1.88/1.90	2.15

Table 2
Measuring the effects of the $\mathsf{abstract}_{\lhd_{pt}+gen}$

The goal of Table 2 is to study the practical benefits that can be obtained by using the new abstraction operator $\mathsf{abstract}_{\unlhd_{pt}+gen_{\circledcirc}}$ proposed in Section 6. As in Table 1, for each specialization approach we show four columns, with the same meaning as before. However, in this case, rather than the absolute data we show just the gains obtained w.r.t. the behavior of $\unlhd_{pt}$, which is shown in absolute terms in Table 1. We have two groups of columns, labeled as $\mathsf{abstract}_{\unlhd_{pt}+gen_{\equiv}}$ and $\mathsf{abstract}_{\unlhd_{pt}+gen_{\approx}}$, each of them shows the gains of using respectively such abstraction operator when compared to using $\unlhd_{pt}$.

As it can be seen, the new global control never introduces relevant overhead. Furthermore, in most cases it introduces relevant speedups, which go as high as 1.5 for the case of $\mathsf{abstract}_{\unlhd_{pt}+gen_{\equiv}}$ and 1.86 in the case of $\mathsf{abstract}_{\unlhd_{pt}+gen_{\approx}}$.

8 Conclusions

In this paper we have studied new mechanisms for achieving "quality" decompilation from Java Bytecode to Prolog while at the same time ensuring termination of the partial evaluation process by using a state-of-the-art *online* partial evaluator. In addition to improving the quality of the residual programs, the techniques we propose provide important efficiency gains during partial evaluation. In particular, we use *partial evaluation types* to provide safe approximations of the values which the arguments of predicates can take during partial evaluation time. Such partial evaluation types are then used by the partial evaluator in order to steer the specialization process, both at the local and global control levels. Besides, we present novel techniques to control the *polyvariance* of the PE process, i.e., to avoid having too many (redundant) specialized versions of some predicates. As we have showed in our experiments, both proposals improve not only the effectiveness but also the efficiency of the decompilation process which, at the same time, widens the class of programs that can be handled by using our interpretative approach. It remains as future work to improve the precision of our techniques to achieve effective decompilation of recursive procedures [7]. To do this, we plan to use more advanced PE techniques [21] which integrate abstract interpretation.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905

MOBIUS project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, LNCS. Springer-Verlag, 2007. To appear.

[2] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, April 2006.

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.13). Technical report, School of Computer Science (UPM), 2006. Available at `http://www.ciaohome.org`.

[4] G. Puebla et al. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, 3049 in LNCS, pages 234–261. August 2004.

[5] Yoshihiko Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems - Proceedings*, pages 1–8, Tokyo, Japan, 1988.

[6] J. Gallagher and D. de Waal. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of ICLP'94*, pages 599–613. MIT Press, 1994.

[7] J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *HOSC*, 14(2,3):143–172, 2001.

[8] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.

[9] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Partial Evaluation Types for Improving the Decompilation of Java Bytecode to Prolog. Technical Report CLIP1/2007.0, School of Computer Science, UPM, February 2007.

[10] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[11] N. D. Jones. Partial evaluation, self-application and types. In *Proc. of ICALP'90*, volume 443 of *LNCS*, pages 639–659. Springer, 1990.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.

[13] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[14] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[15] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[16] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[17] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[18] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd Ext. Ed., 1987.

[19] D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

[20] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.

[21] G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *Proc. of SAS'06*, number 4134 in LNCS. Springer, 2006.

[22] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta programming. In *Proc. IFIP '86*, pages 415–420. North-Holland, 1986.

[23] W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for mercury. In *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 189–232. Springer, 2004.